Re-engineering Legacy Mission Scientific Software

Charles D. Norton[1] and Viktor K. Decyk[1,2]

[1]*Jet Propulsion Laboratory*
California Institute of Technology
MS 168-522, 4800 Oak Grove Drive, Pasadena, CA 91109-8099

[2]*University of California at Los Angeles*
Department of Physics and Astronomy
Los Angeles, CA 90095-1547

Contact Author: Charles.D.Norton@jpl.nasa.gov

JPL missions are constantly evolving, increasing the demands imposed on critical software systems and applications. This ambition requires more modern, flexible, reusable, and extensible component-based software that does not abandon the production applications required for success.

Legacy software has great value since it is generally well debugged, produces results that are trusted, and is actively meeting end-user goals. The amount of hidden expert knowledge embedded in such software can be significant making its preservation important. Nevertheless, legacy software has limitations. It can be difficult to extend, modify, and it does not support collaborative development very well. This can impede the ability to meet new and expanded mission goals as timelines and budgets become tighter. One approach to this problem is to rewrite the software from scratch, but this may introduce more serious costs. In particular, developing new verification and validation tests can be expensive. Also, ensuring that the legacy code was faithfully rewritten, regardless of the programming language applied, cannot always be guaranteed.

Generally, if the functionality of the legacy software is sound, it can be wrapped in a modern interface where the original code is mostly unmodified. The idea of wrapping code means that the original legacy software is preserved while a new layer of software is introduced to separate the old software from the new software. The wrapper provides the best means of retaining the functionality of the legacy software investment while providing a more flexible context from which new software, based on modern concepts, can be introduced.

There are many benefits to this approach:

1. Software remains in productive use while applications are modernized.
2. Avoids costly and potentially harmful software rewrites.
3. Promotes collaborative development while resolving organization problems exhibited in older codes.
4. Re-engineering occurs more quickly than rewriting, while preserving verification and validation tests, especially when the original programmers are involved.
5. Old bugs are uncovered.

We have developed a step-by-step process that allows software to be modernized, while also improving its quality. The application remains in productive use during the entire process. As much as possible, we do not modify the original subroutines, but rather incorporate the modern features in interface (wrapper) libraries. Our methodology is based on Fortran 90/95,

because it has the modern features we desire, while still maintaining backward compatibility. We make use of Fortran 90 language features such as modules, derived types, and dynamic array objects. Embedding the older code inside the interface libraries encapsulates the implementation details of the legacy code, while adding dynamic features and additional safety checks. This also enables multiple authors to work on pieces of the code without interfering with one another and better reflects the problem domain. The new code can evolve toward an object-oriented design, if that is desired. Once the new superstructure works correctly, there is always the option of replacing individual pieces of the legacy code.

We will describe how our methodology has been successfully applied to the Modeling and Analysis for Controlled Optical Systems (MACOS) software important to NASA's Next Generation Space Telescope Project. This software, developed by the Optical Systems Modeling Group at JPL, provides powerful optical analysis tools and a unique capability for system-level design and analysis tasks. The software has been very useful for numerous projects, including diagnosis of the Hubble Space Telescope.

MACOS is written primarily in Fortran 77 and it interoperates with Matlab, PGPlot, and FFTw. There is also a subroutine library called SMACOS based on MACOS. Previous efforts to rewrite the software completely in C++ (to meet new objectives) were abandoned primarily because the new code did not perform as desired, and the designers are more fluent in Fortran. The objectives of the designers were to achieve Fortran 90/95 standard compliance, dynamic memory support, and to identify and correct subtle bugs during the process. It was important that the software remain in use during this effort.

In the presentation our methodology will be described and the software engineering issues involved in modernizing MACOS will be presented. We will also provide statistics on the re-engineering effort and describe how software tools could help in partial automation of this work.